Development of Component Applications for the DESI Online System

UNDERGRADUATE THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Bachelor of Science in the Undergraduate School of The Ohio State University

By

Lucas Wayne Beaufore

Undergraduate Program in Engineering Physics

The Ohio State University

2016

Project Advisor:

Professor Klaus Honscheid, Department of Physics

Abstract

The focus of this thesis project is on the development of applications for use in the DESI Online System, or DOS. DOS is the online system used for the control and management of the Dark Energy Spectroscopic Instrument (DESI). The DESI project will make measurements of the spectra of galaxies and quasars in order to provide data illuminating the nature of dark energy. The control system for this instrument is still in the development stage and must be finished in time for the start of the survey in 2018. As such, this project concerns the creation of the software applications needed to control the individual components of DESI, testing these applications through the use of simulators both provided by the component teams and developed alongside the applications, and finally integrating the applications into the full system. The three primary applications whose development is presented in this thesis are the Telescope Control System Interface, the Spectrograph Control application, and the Cryostat Reader application. These applications, which have been developed primarily in the Python language, will be vital to the operation of the DESI survey.

Acknowledgments

I would like to thank all the teachers, mentors, advisors, friends, and family that supported me throughout my undergraduate education. Thanks are due in particular to Professor Klaus Honscheid and Ann Elliott, for the vast amount of help and guidance they have given me throughout my undergraduate education and during the pursuit of this thesis project. Thank you to Doug Morgan, Kaeli Hughes, and Garrett Merz for taking the time to provide feedback on early drafts of this thesis. Thank you as well to the OSU Department of Physics, OSU Department of Engineering, the Dark Energy Survey Collaboration, and the Dark Energy Spectroscopic Instrument Collaboration.

Table of Contents

List of Figures

Chapter 1: Introduction and Background

1.1 Scientific Background

One of the foremost questions in physics research today is that of the nature of the universe's accelerating expansion. Detection of this phenomenon was first published in 1998 by the High-Z Supernova search team and in 1999 by the Supernova Cosmology Project. The expansion was detected through measurements of the redshifts of Ia Supernovae, which are also known as "standard candles" due to the expectation that they occur with a consistent luminosity [1,2]. There are three leading hypotheses as to the cause of this accelerated expansion. The first is that there is a modification required in the theory of General Relativity for cosmological scales. The second is that the expansion is driven by a hypothetical form of energy that has negative pressure, which would not be due to any particles known or otherwise, called dark energy and would constitute about 68% of the observable Universe's energy density. The third is a "cosmological constant" which would act as a static form of dark energy [3].

1.2 DESI Survey Goals

According to its stated plan, the Dark Energy Spectroscopic Instrument (DESI) [3] is broadly designed to investigate the composition of the Universe at large as well as

the nature of space-time. This investigation will include working to establish which of the previously stated hypotheses is most likely responsible for the Universe's accelerating expansion and constraining models of primordial inflation [3]. This will be done by constructing a 3D map of the universe from precise measurements of the spectra and redshifts from more than 20 million galaxies. This map, unprecedented in volume, will allow for the expansion history of the universe to be charted through measurements of baryon acoustic oscillations and for the growth of structure to be examined via red-shift space distortion measurements [4].

DESI's primary measurement is of baryon acoustic oscillations. Baryon acoustic oscillations (BAOs) are acoustic waves that began as small overdensities in the electron-photon plasma of the early universe that left an imprint on the distribution of matter after recombination (when the plasma turned into neutral atoms). This occurred 380,000 years after the Big Bang. This pattern has the same source as that of the cosmic microwave background (CMB), this connection can be seen in Figure 1. BAOs appear in the distribution of all matter in the Universe, and DESI will examine the positions and spectra of galaxies in order to measure these oscillations [3]. By examining the pattern this leaves behind as a function of the time since the Big Bang, which is determined by the redshift of the galaxies within the pattern, information about the expansion history of the universe can be deduced [4].

DESI will result in independently useful data, but will have the added value of complementing the results of other surveys. Cosmological measurements will be made by cross-correlating data taken from Planck experiment and future CMB experiments that

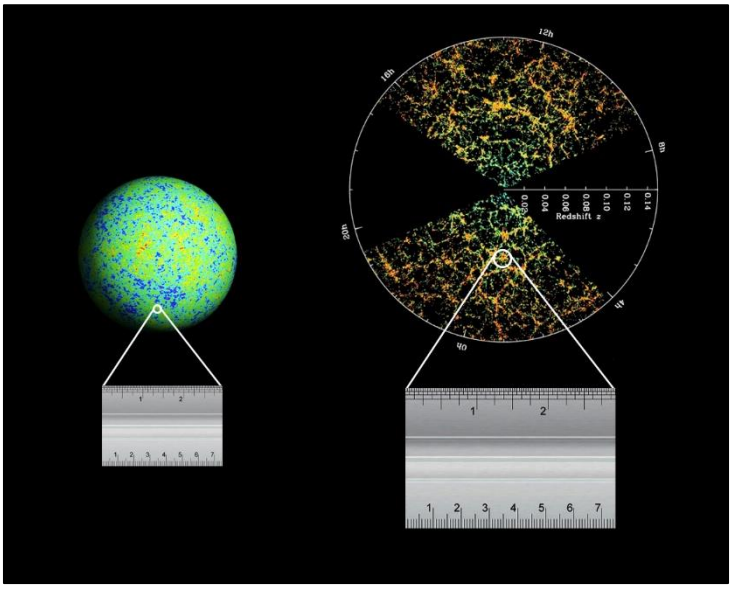

Figure 1: Cosmic Microwave Background and BAO [5]

would not have been possible from the individual data sets [3]. Furthermore, both the Dark Energy Survey (DES) and Large Synoptic Survey Telescope (LSST) have a few thousand square degrees of overlap in survey area with DESI. Multiple cosmological tests, either using overlapping survey area or techniques independent of it, can be performed by examining other data sets alongside DESI's. Finally, the *Euclid* mission being undertaken by the European Space Agency in 2020 and the planned *WFIRST-AFTA* NASA mission will provide an opportunities for complimentary measurements when combined with DESI data, primarily investigations into galaxy-galaxy weak lensing [3].

1.3 Instrument

The DESI instrument was designed to meet the survey's science and operational requirements, which include (but are not limited to) the following [3,6]:

- The DESI survey must cover at least 9000 square degrees of sky, with a maximum practical area of 14,000 square degrees.
- The survey must reach 30 million cosmological targets.
- The instrument should fit about 700 fibers to each square degree of sky.
- The instrument's field of view should be 7 square degrees or more.
- The spectral range of the survey is 360-980 nm.
- The target galaxy density is 2900 per degrees squared with an expected 1800 successful redshifts per degree squared.

These requirements necessitate a high throughput spectrograph that will observe thousands of target spectra per exposure. As such, the system was designed to maximize the throughput from beginning to end.

The DESI instrument will be installed on the Mayall 4-m telescope at Kitt Peak National Observatory (KPNO) which will provide a 3.2-degree diameter field of view [3]. The focal plane will support 5,000 robotically positioned fiber optic cables that will transport the photons from the focal plane to the waiting spectrographs. This large number of fibers will allow for more measurements per exposure, thereby fitting many measurements into a finite survey duration. These robotic positioners will allow for a reconfiguring of the focal plane in 120 seconds (with a goal of 60 seconds) to 5 μm RMS accuracy. This reconfiguring of the focal plane will run in parallel with the telescope slewing to the next target position. The fiber positioners will move to approximate positions during the slew, and then correct to exact positions after the telescope is in place. In addition to these fiber positioners, ten guide, focus, and alignment (GFA)

sensors will be placed in and near the focal plane [3]. These ten sensors will be identical CCD cameras. The guide cameras will be placed in focus and are used for tracking known stars in order to keep the focal plane aligned with the targets. The focus and alignment sensors are placed with dual filters 1.5 mm above and below the focal plane in order to extract out of focus images of stars for analysis. Information from the GFA is used by the six axis hexapod to position the focal plane and optical corrector barrel [3]. The light from the focal plane is transferred to multiple spectrographs via fiber optic cables. The spectrographs are read out in parallel with the telescope slew and focal plane reconfiguring.

1.4 Instrument Control System and DOS

The Instrument Control System (ICS) is responsible for all of the control and monitoring functions required to keep the DESI instrument operating. The software responsible for these functions has been adapted from the Dark Energy Survey's (DES) readout and control system architecture, SISPI. This is due to similarities in the surveys, including the fact that the Mayall telescope being used for DESI is the identical sister telescope to the Blanco telescope used for DES. The updated design of the online system forms the DESI Online System, or DOS. Components such as the dynamic exposure time calculator, the real-time data quality assessment, and complex algorithms to convert on-sky target coordinates to fiber positions on the focal plane were adapted from the SDSS-III/BOSS online system [7]. The ICS is designed to meet the survey's requirements for successful operation including being able to accommodate an exposure timetable of less

than 120 seconds between exposures and a goal of 60 seconds between exposures (as previously mentioned), to be able to withstand the failure of a single computer or drive excluding dedicated hardware interfaces, to provide user interfaces, to provide communication paths between subcomponents of the system, to facilitate data flow, and more [7].

This project is concerned with the development of important elements of the DESI Online System as part of the overall DESI project. This thesis will provide insight into the design and performance of these elements under this project.

Chapter 2: Architecture of the DESI Online System

2.1 Architecture

The DOS architecture is based on that of the Dark Energy Survey's online system, along with the set requirements for the ICS. DOS is responsible for the management of dataflow, the controlling of the instrument, the monitoring of the system, and providing a user interface [8]. DOS manages dataflow by reading data out of the spectrographs, converting that data to the FITS file format while inserting the necessary DESI keywords, and storing those images to a disk archive. DOS is not only responsible for the sequence of exposures taken; it also provides the high level software control for the entire instrument and its sub-components. DOS oversees the system by continuously monitoring operational parameters for the instrument and storing these values in a database, by monitoring environmental information from the Mayall telescope's systems and storing these values in a database as well, by providing interfaces to this database, and by creating alarms and errors that both alert operators and are archived. Finally, DOS provides a Graphical User Interface (GUI) using the Model-View-Controller design pattern to give access to the many systems described above [7].

DOS's architecture is adapted from the architecture of DES's SISPI online system. The schematic diagrams for SISPI and DOS are shown in Figure 2 and Figure 3,

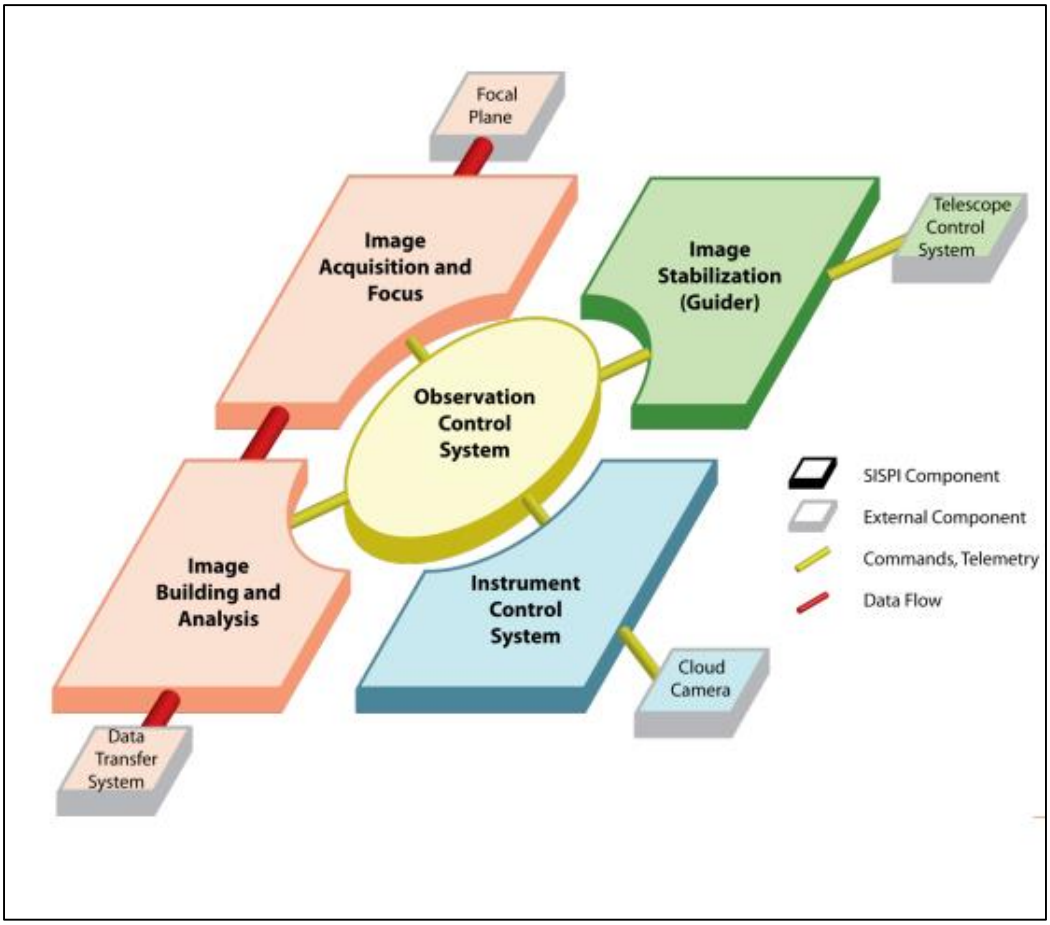

Figure 2: SISPI Schematic View [8]

respectively. There are notable differences between the two systems including the groupings of the focal plane control and the guider, as well as changes in which instrument specific systems are included, such as the spectrographs and fiber positioners in DESI's system. However, clear structural similarities such as the central Observation Control System (OCS), the modular nature of the control scheme, and the separation of the dataflow telemetry and commands are present.

DOS's schematic view shows the OCS at the center of the structure, where it is responsible for overseeing all aspects of DESI's complex exposure sequence. It accepts exposure requests from outside the system which include all necessary information such

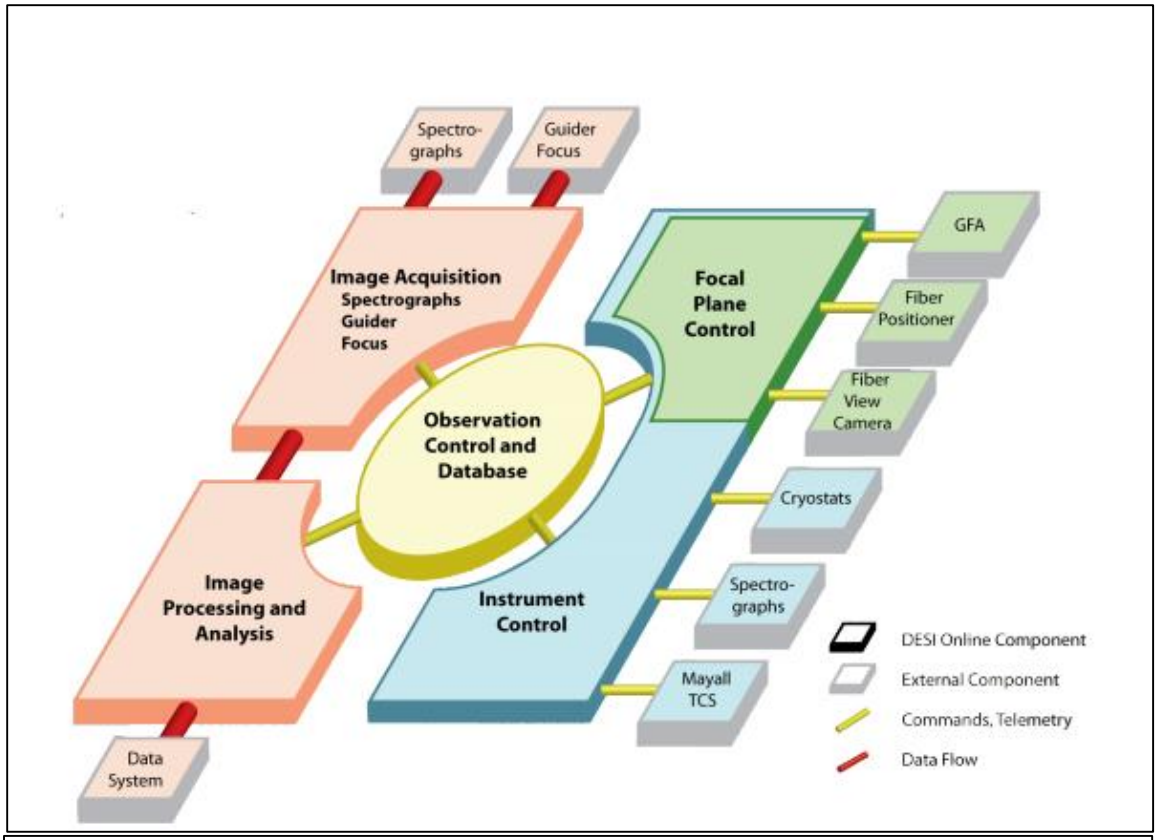

Figure 3: DOS Schematic View [8]

as fiber positions and guide stars. The OCS then coordinates with the various subsystems to align the instrument, take the exposure, and read out the data. The system implements a pipelined architecture that allows data readout to run in parallel with configuring the system for the next exposure [7]. Various subsystems can be seen in DOS's schematic; this thesis project concerned those that can be seen under Instrument Control (in blue). These are the primary focus of development under this project.

DOS's software is primarily developed in Python 3, updated from SISPI's use of Python 2. DOS is constructed to be modular and application focused, thereby allowing for development and testing of subsystems independently. This in turn has the advantage of being able to create "instances" of DOS with different configurations of applications,

depending on what is necessary and what is developed at the time. DOS is a distributed system, running on many computers and controllers throughout the instrument and its surrounding systems. These things are made possible by the Application Framework, the software model used by DOS.

2.2 Application Framework

The Application Framework is built on the Python Remote Object (PYRO) library. The PYRO library allows for communication between Python applications over a network connection by allowing objects to be directly accessible even when located in a different application, hence "PYthon Remote Object". DOS wraps the functionality of PYRO in two different software layers [7]. The first of these is used for sending command and is called Python Messaging Layer, or PML. PML uses a Client-Server design pattern to allow one application to make a function call to another, either to prompt action or to make a request for information. The second is used for sharing data in the form of a Publisher-Subscriber pattern. The data is shared over "Shared Variables", variables in python that can either publish updates or subscribe to them. The data in these variables is accessible to any applications in the DOS instance with a connection to the Shared Variable Engine, or SVE. The SVE is responsible for keeping track of the Shared Variables and distributing updates from publishing variables to the subscribed ones. This is used to provide access to telemetry data for all other applications that may need it.

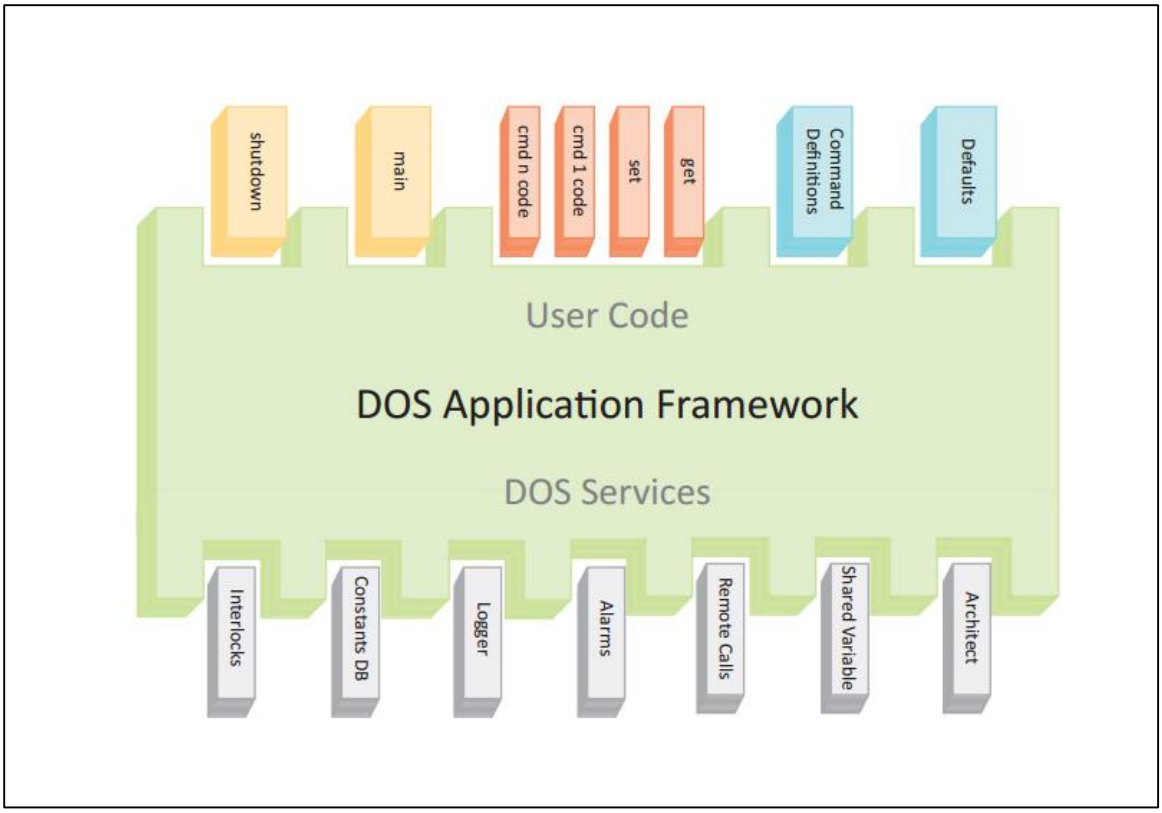

Figure 4: Application Framework Outline [7]

The Application Framework is encapsulated in a Python class that allows for the development of a software application within the same basic structure of all DOS applications, giving it access to DOS services as well as allowing it to be added to an instance of DOS with ease. The model of an application within the DOS Application Framework can be seen in Figure 4.

Developing applications within the Framework allows for easy configuration of the system. The Framework allows a newly starting application to connect to the previously running applications in the instance automatically. A startup system called the Architect has been developed that allows for a complex instance of DOS to be created using command line arguments and configure files of the ".ini" format to specify what

applications should be started as well as configuration parameters to start them with. The procedures for shutting down an application are also handled centrally by the Framework. This allows an application to exit while allowing the rest of the instance to continue running [7].

When a DOS application is built within the Application Framework it first inherits the Application class, which provides access to the DOS services that can be seen in the bottom row of Figure 4, including those previously mentioned. Within the structure laid out by the Application class, the developer then provides the user code that forms the primary functionality of the application (top row of Figure 4). First, the developer can specify “Defaults”, or default values within the program, that can be overridden at startup by either a configuring “.ini” file or command line arguments. Next, the commands that will be accessible to others via the online system are specified so that the Framework can make them available. If necessary, the developer can override the main function and/or shutdown function to control what is done during the run loop to customize the exit handlers. All DOS applications should have a configure function to set the application to an initial ready state. Finally, individual functions within the class are developed. Functions that are accessible across the online system are specified as commands. Those commands must accept arguments of arbitrary number and type which are then parsed by the dos_parser utility.

One of the primary purposes of the Application Framework is to provide the online system with a standardized, modular, and accessible interface to the many elements of the control system which vary widely in terms of hardware, programming

languages, and communication paths. When a DOS application is designed to be a point of communication with a subsystem, it is responsible for wrapping the subsystem's functionality within its remotely accessible commands, its Shared Variables, and its alarms. By doing so, it takes a complex and diverse set of subsystems and makes them easily accessible and configurable to the higher levels of the online system.

Chapter 3: DOS Applications

The primary purpose of this project was the development of DOS applications for use in instrument control. The three most important applications developed were the Telescope Control System (TCS) Interface Application, the Spectrograph Controller application, and the Cryostat Reader application. The development and testing processes for these three applications are presented below.

3.1 Telescope Control System Interface

The TCS is responsible for the control of the telescope itself, the calibration lamps, and the environmental parameters inside the dome. Therefore, the TCS requires an interfacing application to expose this functionality to the online system. This application is known as the TCS Interface, and its initial development is the first part of this project.

The Mayall TCS is a software control program that can be interacted with over an Ethernet socket. This is developed independently from the DOS application "TCS Interface", and so the TCS Interface has been programmed to match the existing and planned functionality for the TCS. Since the Mayall telescope is the identical sister telescope to the Blanco telescope used for the Dark Energy Survey, the TCS Interface was adapted from the equivalent application in SISPI. This was done under the expectation that the two projects' Telescope Control Systems would be reasonably

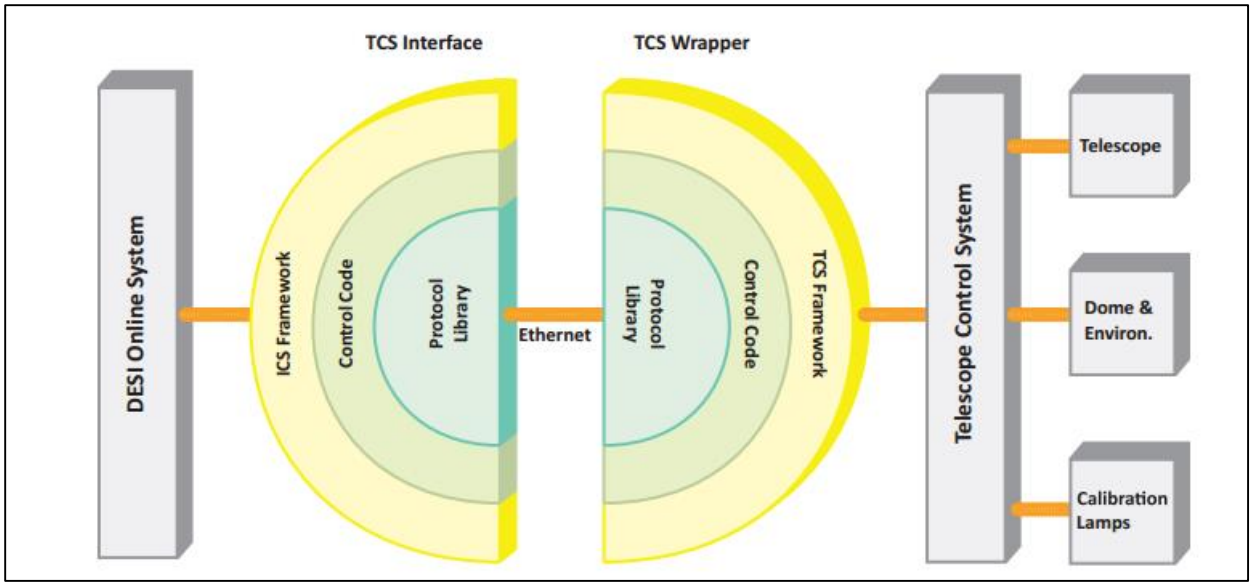

Figure 5: TCS Block Diagram [9]

similar. The structure of the interaction between the DOS application and the TCS can be seen in Figure 5.

The protocol library, represented by the inner semicircles in Figure 5, is defined in an "Implementation Document" [10]. The control code and ICS framework surrounding it represents the TCS Interface application. The communication over the Ethernet socket is always initiated by the DOS side of the connection. All communication is formatted as ASCII strings containing key-value pairs, for both directions. All commands are responded to immediately and all responses include state information ("DONE", "ACTIVE", or "ERROR") followed by any values requested. The TCS Interface wraps all of this communication in remotely callable commands and Shared Variables for use by the rest of the system.

The majority of the commands in the TCS Interface are concerned with the movement of the telescope. The basic command for this is "move_telescope" which

accepts coordinates for the telescope position either relative to a location on the sky, to the earth, or to the telescope's current position. Once this command is processed, the TCS Interface monitors the telescope's process, and reports back when the slew is determined to be either a success or a failure. Other commands concerning the telescope's movement are the "whitespot" and "zenith" commands which send the telescope to known positions, the "stop" and "abort_slew" commands which cease any telescope motion and cancel a running slew to a new position respectively, the "track" command which requests that the telescope start tracking the movement of the sky, and the "guide_correction" command that is responsible for updating a telescope with corrections to its current movement.

The TCS Interface supports commands beyond ordering telescope movements, namely being responsible for updating the TCS with the current state of the six axis hexapod (for use in the TCS's pointing model) and controlling the calibration lamps. The "hexapod_position" command takes six floating point values that describe the x, y, z, tip, tilt, and rotation of the hexapod. The "calib" command controls which lamps are on and what brightness they are set to. The application also supports "get" and "set" commands that provide access to information both from the TCS and from the application itself.

Finally, the TCS Interface is responsible for regularly updating Shared Variables with information about the state of the TCS and the application. The three primary Shared Variables for this purpose are "INFOT", "INFOE", and "INFOC" which contain telemetry, environmental, and calibration information, respectively. These three Shared Variables are updated periodically by independently running update threads that are

Figure 6: The Mayall Telescope [11]

started when the "configure" command is first run. The information published on these Shared Variables is used in another program to update the experiment's database.

To facilitate the development of the TCS Interface, two simulators have been used to mimic the TCS's functionality. The first simulator was created alongside the TCS Interface as part of this project. Developed in Python, the simulator contains a Python socketserver that communicates through the ASCII protocol defined in the Implementation Document [10]. The socketserver has a function that handles requests by parsing the incoming ASCII into the requested command and arguments. These values are used to interact with a TCS simulator object that mimics a simplified TCS. The simulator includes internal time delays to mimic the slewing of the telescope, position calculations using the PyEphem astronomy package, slowly drifting environmental data, and simple tracking of the hexapod and calibration lamp's statuses. This simulator was

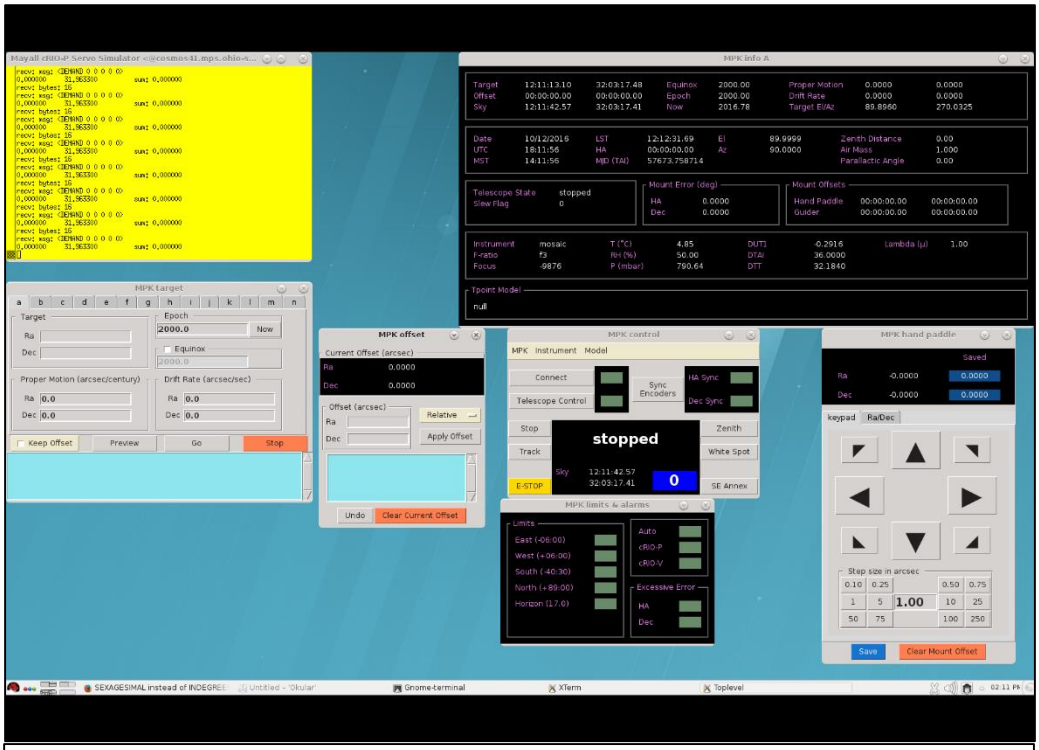

Figure 7: The TCS Simulator User Interface

used until the official simulator was delivered by the team responsible for the real TCS system at a later date. This second simulator uses identical communication to that of the first simulator (since both mimic that of the final system), but includes physical details of the telescope as well as safety systems that were not accounted for when calculating responses in the first simulator. The simulator also includes a graphical interface so that the behavior of the TCS can be compared against what is intended by the TCS Interface. A subset of this interface can be seen in Figure 7.

Previous to the development of the TCS Interface, most applications had been spot checked by hand to make sure they functioned as intended. In order to verify that the applications work consistently, including in border cases, unit testing was introduced as a development tool throughout this project. Unit testing is a way of automatically running a

series of tests, usually involving a call to a command and an analysis of the response. If the command and response behave as expected, the test passes; if not, it fails. By creating a battery of unit tests that probe a variety of commands to test for their behavior in both common usage and border cases, the integrity of the application can be verified quickly after each change. This way, changes in the performance of one part of the application can be checked when edits are made to an unrelated section. These tests are implemented using Python's "unittest" library, and can be used to verify that TCS Interface is fully operational as development continues.

3.2 Spectrograph Controller

The primary measurement device of DESI is the spectrograph array fed by the fiber optics transporting light from the focal plane of the instrument. There will be 10 spectrographs in DESI, each with a red, blue, and near infrared CCD camera for taking data. These spectrographs are equipped with network-enabled controllers that interface with the hardware and control the operation of the spectrographs. Each network-enabled controller includes a Python software library that allows for external control of the individual components of the spectrograph while protecting from any incorrect or illegal commands. The DOS application will run on the network-enabled controller and communicate over the network with the rest of the system. This structure is replicated for each spectrograph, as can be seen in Figure 8. The creation of the DOS application for this system is the second portion of this thesis project.

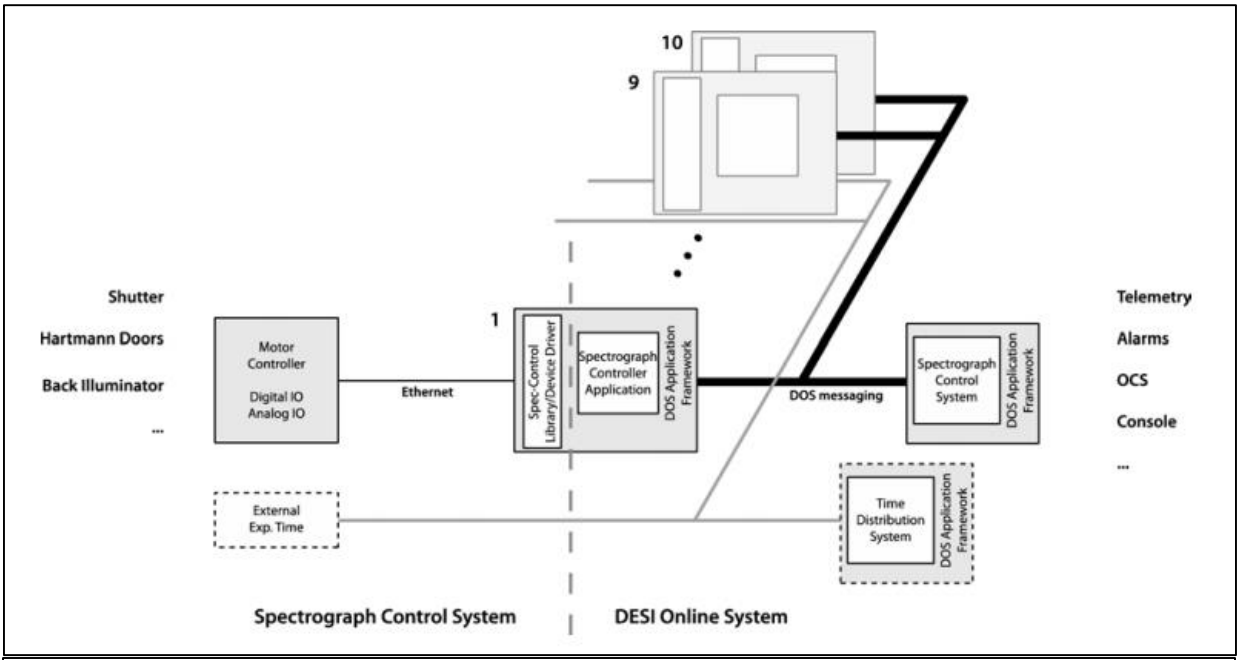

Figure 8: Spectrograph Control System Interface [12]

The DOS application for this subsystem, SpectroControl, is responsible for integrating the spectrograph control library into DOS. SpectroControl, in addition to the common commands present in DOS application such as “configure”, “get”, and “set”, provides commands that allow the observation control system to access direct control of the components. These commands include changing the power settings for the shutters and the Hartmann doors in the spectrograph, opening and closing these shutters and doors, and inflating and deflating the shutters’ pneumatic seals. There are also commands that prepare the spectrograph for an exposure or for illumination, either through use of a given preparation command in the library or by setting the components to their required states. The DOS application includes functions for illuminating and exposing the spectrograph, including the ability to pause and resume an exposure. Finally, the application monitors the sensors for the spectrograph hardware, recording the

measurements in Shared Variables and raising alarms if any of the values leave a safe range.

Like the TCS Interface, the SpectroControl application requires a simulator in order to test the functionality of the DOS application prior to deploying the code on a physical system. However, unlike the TCS, the spectrograph library has no official simulator provided by the group developing it. Therefore a more complete simulator for the spectrograph hardware was developed as a part of this project.

Since the spectrograph control library is a Python module, the simulator was developed as a Python module as well. It contains a SpectroController class that has functions used for retrieving data and sending commands, identical to the true module. The SpectroController class stores the state of the system in a series of dictionaries which are updated to reflect any changes. The sensor data values for the spectrograph are given initial values which are updated periodically in an independent thread. These updates take the form of a random value added to the current one, with the final range restricted to within believable physical limits. The ability to set these values and to lock them into place is also provided to allow for testing of specific border cases. This is implemented as PML accessible commands so that the test bench can change the values independently from the SpectroControl application.

Each simulation function in the library checks the state of the system before running and throws a runtime error if such a call would result in an illegal or unphysical action, such as moving an unpowered shutter. If no such error occurs, the function then waits for a time mimicking the physical delay time of the action before updating the

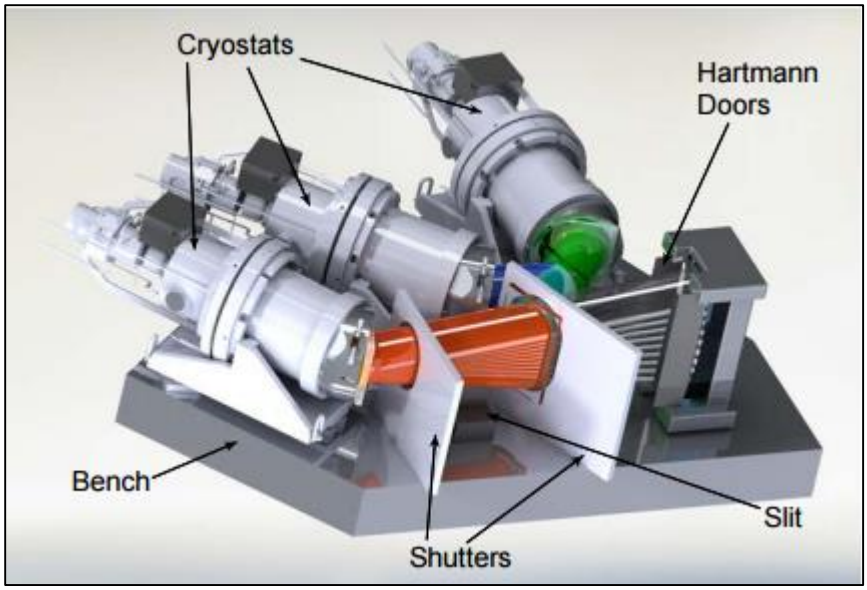

Figure 9: Spectrograph Hardware CAD Rendering [3]

simulator state and returning. If the simulator is called to begin an exposure or an illumination, a new thread is started that is responsible for updating the state of the simulator, replicating the exposure/illumination process until either the time expires or it receives a stop command.

Due to the necessity of separating errors in the DOS application from errors in the simulator, a series of unit tests have been developed to verify the correct behavior of the simulator. These tests show that the simulator works as expected for both normal use and border cases, and so any errors arising in testing the ongoing development of the SpectroController application will be due to errors in the application itself.

3.3 Cryostat Reader

Each of the spectrographs is equipped with three cryostats, devices designed to maintain a constant low temperature, one for each camera. This setup can be observed in

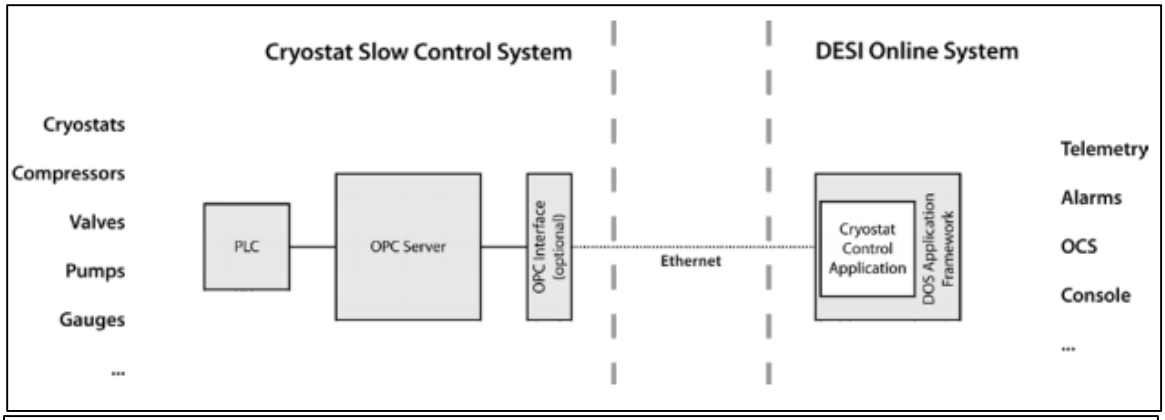

Figure 10: Cryostat Software Block Diagram [13]

Figure 9. Although a DOS application will not be dedicated to controlling the cryostats, there is an application dedicated to monitoring environmental and state information published by the cryostats. This application is the CryostatReader, and its creation is the third part of this project. Unlike the prior two applications, which involved controlling hardware through a simple path of communication, the CryostatReader navigates a read-only OPC server containing the state information about each cryostat in leaf nodes and publishes that information on Shared Variables. The data is then directed to eleven different tables (one per spectrograph and one general table) in the telemetry database which store the information. The CryostatReader is also responsible for raising alarms in DOS in response to changes in the cryostats. The relationship between the cryostat's OPC server and the DOS application CryostatReader is shown in Figure 10. The optional OPC Interface is unused.

The box containing the Cryostat monitoring hardware was shipped to OSU to aid in the development of the DOS interface. It contained the power system, which had to be converted from French to American electrical standards, and the computer containing the server. The French team responsible for the hardware had configured the server to

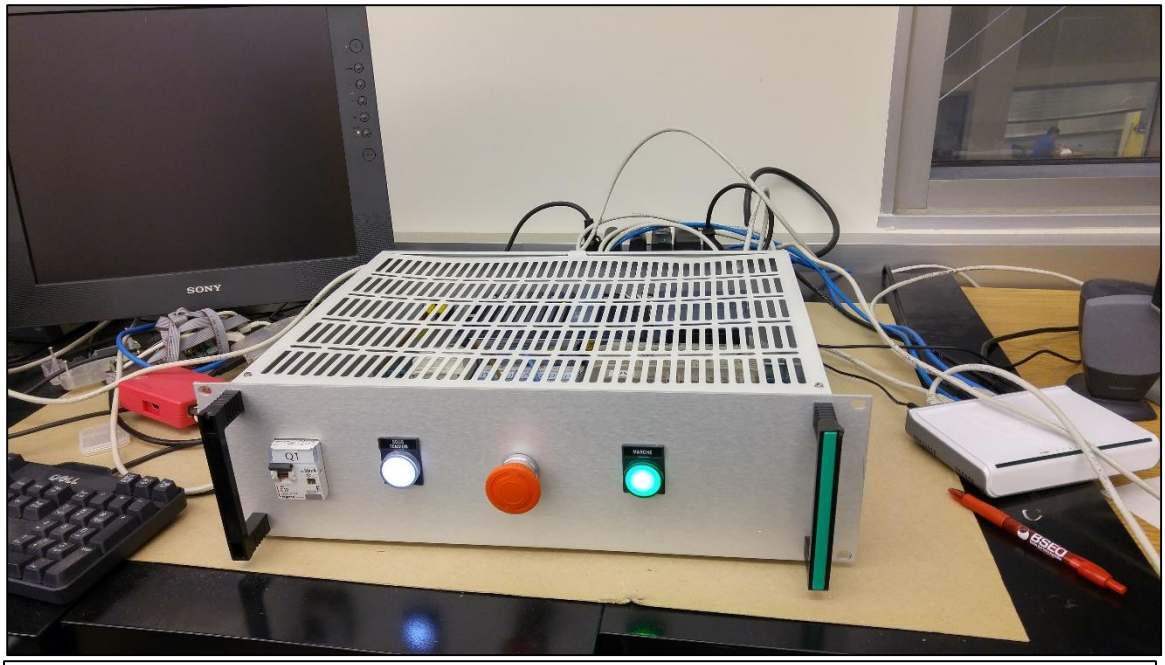

Figure 11: Cryostat Server Box

contain the data fields that would be present in the final version of the system, and to fill these fields with changing values for the purpose of testing. This box can be seen in Figure 11. The first step was of creating the CryostatReader was to uncover the server file structure as well as the names and data types of recorded values within the system. This was done by using the UaExpert GUI client provided by Unified Automation (Figure 12). With the server structure established, the next step was to create the DOS application to interface with it.

In order to navigate the OPC server a suitable OPC client Python library had to be found. Locating such a library proved to be more challenging than expected. Libraries tried included the OpenOPC, PyUAF, and open62451 libraries. All either proved to be unable to connect to the server or did not integrate well with the existing structure.

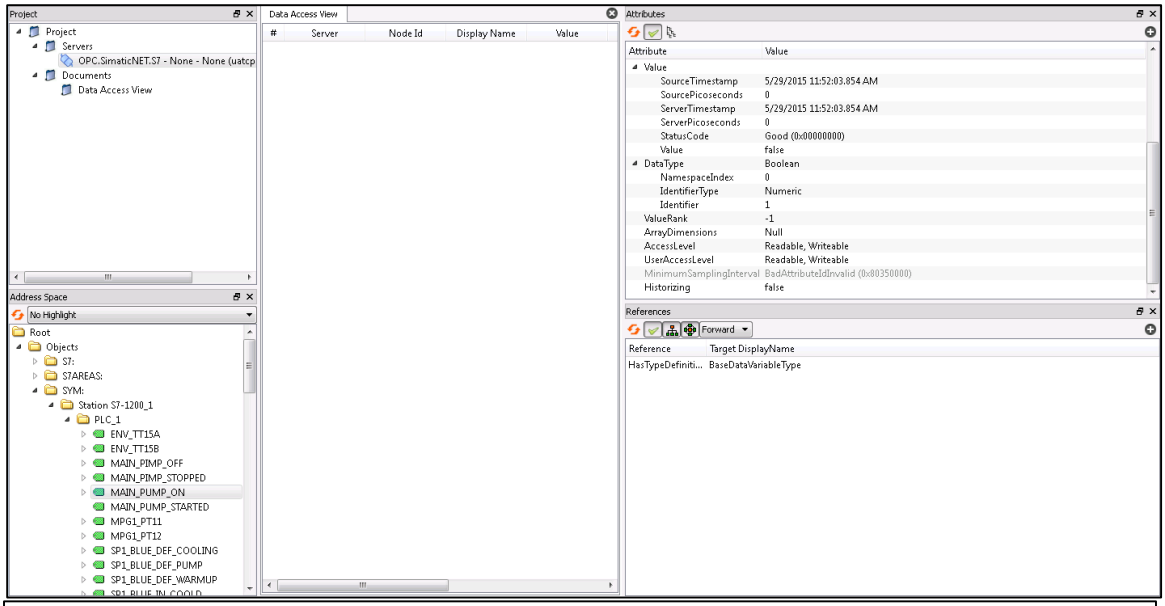

Figure 12: UaExpert GUI Client

Finally, the opcua library from FreeOpcUa was used to access the server and the CryostatReader application was developed around this library.

The CryostatReader first navigates to the directory containing all the data nodes for the system. It then examines all the node names to determine how many spectrographs are represented in the server at that time, and sorts the data by spectrograph, including a section for non-spectrograph specific data. It then begins an update cycle of polling all the nodes for their current values, recording the values into a Python dictionary that corresponds to the destination table, and then publishing those dictionaries to a Shared Variable.

The CryostatReader includes an internal simulator that can be used when no actual server is present. Because of this, the application can still be run as part of a test system even if no cryostat or monitoring server is present. The Cryostat reader also

includes the ability to respond to alarm nodes set by the server by raising corresponding alarms in DOS.

Unlike other DOS applications that are fundamentally part of a network of applications in DOS (an “instance” of DOS), the CryostatReader was originally designed to function on its own by not extending the Application Framework. This would allow it to continue to monitor the device without being reliant on any outside influence, and would keep running even when no instance was up. Data was inserted into the database through the use of the psycopg2 library. The most recent revision of the CryostatReader has relaxed this requirement, and the CryostatReader now implements the Application Framework. However, it keeps its ability to run independently of a DOS instance by running in “device mode”. This mode allows for a DOS application to run without a PYRO connection to the rest of the system. However, since the data is now published in Shared Variables rather than being inserted in the database directly using psycopg2, a connection to the database updating program must be established, else the data will be lost.

3.4 Other DOS Applications

Although the TCS Interface, the SpectroControl application, and the CryostatReader formed the focus of this thesis, other DOS applications were developed under this project as well. One such application is the NetSwitch application, which is capable of controlling a network-enabled power switch. This would be used to remotely control what instruments were powered from within the online system. The PLC

application was developed under this project to be used by another team within the collaboration. It is capable of controlling a Programmable Logic Controller from within DOS. This can be used to control a hardware test bench for the equipment from the same instance of DOS running the system itself. This included both the application itself and a simulator for development. Finally, an application for interfacing with a sbig CCD camera for use as a Fiber Photometry Camera (FPC) was developed under this project. The application used the python sbig module to control the state of and extract images from the camera. The images were then stored in the FITS file format, and a header was attached. In addition to the application itself, a series of unit tests were developed to verify functionality.

Chapter 4: Conclusion

4.1 Results/Current State

All three applications developed as a part of this thesis project are currently functioning as intended, and are being updated as the software and hardware surrounding them evolves. DOS is still under development, and development will likely continue past the beginning of the survey in 2018. The TCS Interface has been used to control the Mayall Telescope itself during a test run this year, and was found to be fully functional. Since then, updates have been made to keep the application current with the development being done by the team responsible for the TCS. The SpectroControl application was found to work correctly with the simulator at the end of its development under this project, and has since been updated by other members of the collaboration as part of DOS's ongoing development. It is currently being used to control the spectrograph test stand at Winlight, in France. The CryostatReader has been continuously updated in response to changes made by the team responsible for the cryostat, and has functioned correctly at the end of each of these development cycles. The data reported by the CryostatReader has been verified in the telemetry database, and the alarm functionality was developed and verified as the most recent update under this project. It is also being used to monitor the physical test bench at Winlight. The NetSwitch application is fully

functioning, the FPC application was able to correctly control and extract images from the camera, but much more development is needed before production. Finally, the PLC application was verified to work against the simulator, but the team it was developed for has yet to report its usage or performance.

4.2 Future Developments

The future developments of this project are straightforward: to continue to update the existing applications to match developments both in DOS and in other parts of the collaboration. In addition, the remaining applications must be developed and tested before the survey begins in 2018. The next application expected to be developed as a continuation of this work is the Active Optics System application.

4.3 Conclusion

Under this project, multiple DOS applications essential to the Dark Energy Spectroscopic Instrument survey have been developed and tested. These applications will continue to be updated, and will be used over the course of the DESI survey, which will begin in 2018. The DESI survey will then provide measurements that will help to illuminate the nature of Dark Energy.

References

[1] RIESS A. et al. (1998) Observational evidence from supernovae for an accelerating universe and a cosmological constant. Astron. J. 116, p.1009-1038.

[2] PERLMUTTER S. et al. (The Supernova Cosmology Project) (1999). Measurements of Omega and Lambda from 42 high redshift supernovae. *Astrophysical Journal* 517 (2). p.565–86.

[3] DESI COLLABORATION. (2014) *DESI Conceptual Design Report.* [Online] Available from: http://desi.lbl.gov/wp-content/uploads/2014/04/DESI_CDR_20140827_1135.pdf. [Accessed: 22nd December, 2014]

[4] LEVI M. et al. "The DESI Experiment, a whitepaper for Snowmass 2013". In: ArXiv e-prints (Aug. 2013). arXiv: 1308.0847 [astro-ph.CO].

[5] Ruler-CMB-Today [Digital image]. (n.d.). Retrieved Nov 15, 2016, from http://newscenter.lbl.gov/wp-content/uploads/sites/2/2009/10/ruler-cmb-today.jpg

[6] DESI. DESI Level 1 through Level 3 Requirements, Science Objectives, Survey Data Set, Instrument Technical Requirements. DESI-doc-318. Nov. 2013.

[7] DESI COLLABORATION. "The DESI Experiment Part II: Instrument Design". In: ArXiv e-prints (Oct. 2016). arXiv: 1611.00037 [astro-ph.IM].

[8] HONSCHEID K. (2015) MS DOS An Introduction to the DESI Online System [PDF]. Retrieved from https://desi.lbl.gov/DocDB/cgi-bin/private/RetrieveFile?docid=997;filename=wbs1.7-dos-overview.pdf;version=3

[9] HONSCHEID K. et al. (2016). The DESI instrument control system. *Proc. SPIE 9913, Software and Cyberinfrastructure for Astronomy* IV, 99130P doi:10.1117/12.2229835.

[10] HONSCHEID K. et al. (2015) ICS – Mayall TCS API [PDF]. Retrieved from https://desi.lbl.gov/DocDB/cgi-bin/private/RetrieveFile?docid=1132;filename=DESI-1132-v1-TCS-DOS-API.pdf;version=2

[11] Mayall 4-m telescope [Digital image]. (n.d.). Retrieved November 12, 2016, from http://www.weasner.com/observatories/Kitt_Peak_2010/Kitt_Peak_10_Oct_29_24.jpg

[12] HONSCHEID K. et al. (2015) ICS – Spectrograph Controller ICD [PDF]. Retrieved from https://desi.lbl.gov/DocDB/cgi-bin/private/RetrieveFile?docid=556;filename=DESI-0556-v6-Spectro-ICS-ICD.pdf;version=6

[13] HONSCHEID K. et al. (2015) ICS – Cryostat Control System ICD [PDF]. Retrieved from https://desi.lbl.gov/DocDB/cgi-bin/private/RetrieveFile?docid=558;filename=DESI-0558-v4-Cryo-ICS-ICD.pdf;version=4

Appendix A: Sample DESI Application – NetSwitch.py

This code is provided as an example of the format of a DOS application. It is the shortest application developed under this project, used for controlling a network enabled power strip. The other applications, especially the three most discussed ones in this thesis (TCS Interface, SpectroControl, and CryostatReader), are much larger and are omitted for the sake of space.

```
#!/usr/bin/env python

from DOSlib.application import Application
import DOSlib.discovery as discovery
from DOSlib.monitor import Monitor
import DOSlib.multicom as multicom
import DOSlib.interlock as Interlock
import dlipower
import pycurl
import time
import os
import NetSwitch_module

# Parameters
ns_version = '1.0.0'

# Default constants in case database is not availible
ns_constants = {'outlet_1'  :   'device_1',
                'outlet_2'  :   'device_2',
                'outlet_3'  :   'device_3',
                'outlet_4'  :   'device_4',
                'outlet_5'  :   'device_5',
                'outlet_6'  :   'device_6',
                'outlet_7'  :   'device_7',
                'outlet_8'  :   'device_8'
                }

class NetSwitch(Application):
```

```
    """A class that communicates with the network controlled (power) switch

    Attributes:
        address (str): The address of the network controlled switch

    """
    commands = ['configure', 'run_script', 'control_outlet',
'control_device', 'get_device', 'get_outlet', 'add_outlet_mapping',
'get_state']
    defaults = {
    "switch_address"    :   "desipower.mps.ohio-state.edu",
    "delay_time"        :   0.5
    }

    def init(self):
        """ The init function for MulticomServer.

        Args:
            switch_address  (str, optional):        The address of the
network controlled switch
            uname           (str, optional):        The username to use
with the network controlled switch
            userpassword    (str, optional):        The password to use
with the network controlled switch
            devices         (dict, optional):       A dictionary where the
keys are the outlet number, and the value is the name of the device

        """

        self.uname = os.getenv('NETSWITCH_USER', 'admin')
        self.password = os.getenv('NETSWITCH_PASS', '1234')
        self.addr = self.config['switch_address']
        self.delay_time = self.config['delay_time']
        self.switch = NetSwitch_module.NetPowerSwitch(self.addr,
self.uname, self.password)
        self.device_map = {}
        self.constants_version = 'DEFAULT'
        self.constants = {}
        self._setup_initial_constants()
        c = self.get_constants('net_power_switch')
        if c is not None:
            self.constants['net_power_switch'] = c
        else:
            return "FAILED: Can't load required constants: %s" % const
        self.outlet_svs = {}
        self.last_measurement = {}
        for i in range(8):
            self.outlet_svs[i+1] = self.shared_variable("OUTLET%i"%(i+1))
            self.outlet_svs[i+1].publish()
            self.last_measurement[i+1] = None
        for i in list(self.outlet_svs.keys()):
            val = self.switch.get_state(i)
```

```
            self.outlet_svs[i].write(val)
            self.last_measurement[i] = val
        self.updating = False


    def configure(self,  *args, **kwargs):
        """ Restores the system to a known state. This includes the
constants specified, and uses the
            default set if none are specified. The default state for the
spectrograph itself is that
            set by the underlying systems "initialize" command.

            Args:
                constants (str, optional):  The set of constants to use.

        """
        args, kwargs = util.dos_parser(*args, **kwargs)
        if(len(args)>0):
            constants = args[0]
        if constants is not None  and constants != '':
            if constants.lower() == 'default' or constants.lower() ==
'none':
                self.constants_version = 'DEFAULT'
            elif constants.find(':') == -1:
                return 'FAILED: Invalid constants version %s' %
repr(constants)
            else:
                self.constants_version = constants
        if self.constants_version == 'DEFAULT':
            self._setup_initial_constants()
            self.info('Configuring using default constants...')
        else:
            snapshot, sep, version = self.constants_version.partition(':')

        # Load Constants
        try:
            v = int(version)
            ret_code = self.load_constants(snapshot = snapshot,
snapshot_version = v)
        except:
            v = version
            ret_code = self.load_constants(snapshot = snapshot, tag = v)
        if ret_code == self.FAILED:
            return "FAILED: NetSwitch can't load constants %s" %
self.constants_version
        self.info('Configuring using constants snapshot %s (version/tag
%s)' % (snapshot, str(v)))
        self.constants_version = self.get_constants_version()

        # Get Constants
        for const in ['net_power_switch']:
            c = self.get_constants(const)
```

```
            if c is not None:
                self.constants[const] = c
            else:
                return "FAILED: Can't load required constants: %s" % const
        self.dev_map = {}
        for i in range (1,9):
            dev_map[i] = self.constants['outlet_%i'%i]
        self.device_map = dev_map

    def _setup_initial_constants(self, *args, **kwargs):
        """ Sets up the initial set of constants for the controller. """
        args, kwargs = util.dos_parser(*args, **kwargs)
        self.set_constants('net_power_switch', ns_constants)
        self.constants_version = 'DEFAULT'

    def run_script(self, args, kwargs = util.dos_parser(*args, **kwargs) ):
        """Run the BASIC script that begins at line <line> in the switch.
"""
        args, kwargs = util.dos_parser(*args, **kwargs)
        if (len(args)>0):
            line = args[0]
        else:
            line = 1
        return self.switch.run_script(int(line))

    def control_outlet(self, *args, **kwargs):
        """Run perform an action on the outlet(s) at <outlet_num>. """
        args, kwargs = util.dos_parser(*args, **kwargs)
        self.updating = True
        outlet_num = []
        for i in range(len(args)-1):
            outlet_num.append args[i]
        command = args[-1]
        temp = []
        for item in outlet_num:
            temp.append(int(item))
        outlet_num = temp
        ret_val = self.switch.control_outlet(outlet_num, command)
        if ret_val == 'DONE':
            for item in outlet_num:
                self.last_measurement[int(item)] = command
        self.updating = False
        return ret_val

    def control_device(self, *args, **kwargs):
        """Run perform an action on the outlet attached to the listed
device(s). """
        args, kwargs = util.dos_parser(*args, **kwargs)
        device = []
        for i in range(len(args)-1):
            device.append args[i]
        command = args[-1]
```

```
        self.updating = True
        ret_val = self.switch.control_device(device, command)
        if ret_val == 'DONE':
            for item in device:
                self.last_measurement[int(self.get_outlet(item))] = command
        self.updating = False
        return ret_val

    def get_device(self, *args, **kwargs):
        """Get the device attached to given outlet number."""
        args, kwargs = util.dos_parser(*args, **kwargs)
        return self.switch.get_device(int(args[0]))

    def get_outlet(self, *args, **kwargs):
        """Get the outlet number of attached device. """
        args, kwargs = util.dos_parser(*args, **kwargs)
        return self.switch.get_outlet(args[0])

    def add_outlet_mapping(self, *args, **kwargs):
        """Add outlet-device pairing. List the outlet first as an integer.
"""
        args, kwargs = util.dos_parser(*args, **kwargs)
        outlet = args[0]
        device = args[1]
        return self.switch.add_outlet_mapping(int(outlet), device)

    def get_state(self, *args, **kwargs):
        """Get the state for a given outlet. """
        return self.switch.get_state(int(args[0]))

    def main(self):
        while not self.shutdown_event.isSet():
            if not self.updating:
                for i in list(self.outlet_svs.keys()):
                    val = self.get_state(i)
                    if val != self.last_measurement[i]:
                        self.alarm_warning('Power for outlet %i has been
switched to %s'%(i, val))
                    self.outlet_svs[i].write(val)
                    self.last_measurement[i] = val
                self.sleep(self.delay_time)


if __name__ == "__main__":
    MyApp = NetSwitch()
    MyApp.run()
```